Typed Template Coq – Certified Meta-Programming in Coq

Abhishek Anand, Simon Boulier, Nicolas Tabareau, Matthieu Sozeau

Typed Template Coq

Certified Meta-Programming in Coq

Abhishek Anand
Cornell University
Ithaca, NY, U.S.A.

Simon Boulier
Nicolas Tabareau
Gallinette Project-Team, Inria
Nantes, France

Matthieu Sozeau
Pi.R2 Project-Team, Inria and IRIF
Paris, France

Abstract

Template-Coq[1] is a plugin for Coq, originally implemented by Malecha [7], which provides a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Initially, it was developed for the purpose of writing functions on Coq's AST in Gallina. Recently, its use was extended for the needs of the CertiCoq certified compiler project [2], which uses it as its front-end language and to derive parametricity properties [1], and the work of [5] on extracting Coq terms to a CBV λ-calculus. However, the syntax currently lacks semantics, be it typing semantics or operational semantics, which should reflect, as formal specifications in Coq, the semantics of Coq itself. This is an issue for CertiCoq where both a non-deterministic small step semantics and a deterministic call-by-value big step semantics had to be defined and preserved, without an "official" reference specification to refer to. Our hope with this work is to remedy this situation and provide a formal semantics of Coq's implemented type theory, that can independently be refined and studied. By implementing a (partial) independent checker in Coq, we can also help formalize certified translations from Coq to Coq (Section 3).

ACM Reference Format:
Abhishek Anand, Simon Boulier Nicolas Tabareau, and Matthieu Sozeau. 2017. Typed Template Coq: Certified Meta-Programming in Coq. In *Proceedings of ACM Conference (CoqPL'17).* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Reification of Coq Terms

The reification of Coq terms follows Malecha's original work, where the AST of Coq terms is defined in Coq and an ML plugin allows users to automatically construct reified versions of terms or constants with their references. Currently, the system reifies all Coq terms, including fixpoints and cofixpoints, but the unquoting/reflection mechanism is partial. Apart from the use of OCaml's native arrays and strings in the implementation of Coq, the reified syntax is identical to the `constr` datatype representing syntax in Coq's OCaml code. The reifier can also crawl the environment for the dependencies of a given definition to output a `program`, which is the reification of (a minimial subset of) the environment and the definition's reified body. A typical example is given in figure 1. As can be seen there, declarations of inductive types, with their constructors, and constants with their types and bodies, including lambda abstractions, pattern-matching (tCase) and fixpoint constructs (tFix) are all present.

[1] https://template-coq.github.io/template-coq

CoqPL'17, January 2018, Los Angeles, CA, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
Quote Recursively Definition add_quoted := Nat.add.
Check eq_refl : add_quoted =
  PType "Coq.Init.Datatypes.nat" 0
  (("nat", tSort sSet,
    {| ctors := ("O", tRel 0, 0) :: (("S", tProd nAnon (tRel 0) (tRel 1), 1) :: nil)%list |}) :: nil)
  (PConstr "Coq.Init.Nat.add"
     (tProd (nNamed "n") (tInd (mkInd "Coq.Init.Datatypes.nat" 0))
        (tProd (nNamed "m") (tInd (mkInd "Coq.Init.Datatypes.nat" 0))
           (tInd (mkInd "Coq.Init.Datatypes.nat" 0))))
     (tFix ({| dname := nNamed "add"; dtype := _; dbody :=
     tLambda (nNamed "n") (tInd (mkInd "Coq.Init.Datatypes.nat" 0))
       (tLambda (nNamed "m") (tInd (mkInd "Coq.Init.Datatypes.nat" 0))
         (tCase (mkInd "Coq.Init.Datatypes.nat" 0, 0)
            (tLambda (nNamed "n") (tInd (mkInd "Coq.Init.Datatypes.nat" 0))
               (tInd (mkInd "Coq.Init.Datatypes.nat" 0))) (tRel 1)
      ((0, tRel 0) :: (1, tLambda (nNamed "p") (tInd (mkInd "Coq.Init.Datatypes.nat" 0))
                          (tApp (tConstruct (mkInd "Coq.Init.Datatypes.nat" 0) 1)
                             (tApp (tRel 3) (tRel 0 :: tRel 1 :: nil) :: nil))) :: nil)));
        rarg := 0 |} :: nil)%list 0) (PIn (tConst "Coq.Init.Nat.add"))).
```

Figure 1. Example reification of addition on natural numbers.

Modularity of reification. Template-Coq allows us to write many OCaml plugins, especially meta-theoretic translations of Gallina programs (§3), in Gallina itself. Writing such translations in Gallina is advantageous because it provides an opportunity to prove the correctness of such translations correct in Coq. A drawback, however, is that such Gallina programs run much slower than native OCaml plugins. Template-Coq now allows us to get the best of both worlds. It provides a mechanism to extract any Gallina program written over Template-Coq to OCaml and then run the result as an OCaml plugin. To achieve this, we have functorized our reification function, which is written in OCaml. The reification function can be instantiated differently to support both running inside Coq and after extraction.

For example, we can implement a minimal wrapper for CertiCoq's compiler:

```
Definition threeplusfour := Nat.add 3 4.
CertiCoq Compile threeplusfour.
```

The new command goes through the extracted versions of the compiler to produce a file `threeplusfour.c` that can be compiled by CompCert or gcc and linked to a runtime library to compute the value of the threeplusfour definition.

2 Type Checking Coq in Coq

The verification of Coq in Coq has been a long term goal, tackled notably by Barras [3]. While we would like to ultimately prove metatheoretical results for the system described by our typing rules, our initial goals are slightly different in the sense that we focus on reflecting the actual implementation of Coq terms and providing a comfortable setting to work with these definitions. At the time of writing, we have specified:

1. α-equivalence of terms and a "syntactic" cumulativity test that extends α-equivalence.

2. Typing rules for all constructions except the guard check for fixpoint and productivity of cofixpoints. This includes the basic dependent lambda calculus with lets, global references to inductives and constants, the `match` construct and primitive projections. We do not treat metavariables currently which are absent from kernel terms and require a separate environment for their declarations. Universes are not treated either yet, ongoing work by Y. Forster and A. Anand integrates them in Template-Coq.
3. Untyped conversion and cumulativity, as parallel reduction to the same term.
4. The well-formedness judgment for global declarations, except for the positivity condition of mutual inductive types.

Implementation. To test this specification, we implemented the basic algorithms for type-checking in Coq, that is, we implement type inference: given a context and a term, output its type or produce an error. All the rules of type inference are straightforward except for cumulativity. The cumulativity test is implemented by comparing head normal forms and calling itself recursively in Coq's kernel. We can implement weak-head reduction by mimicking Coq's kernel implementation, which is based on an abstract machine inspired by the KAM.

Of course all of these functions are required to be terminating, so we make them depend on a fuel parameter. This could also be implemented by well-founded recursion on a proof of strong normalization for the input terms, but we found the fuel way easier to deal with as a first step. One of the typechecking errors then becomes `OutOfFuel`.

We are working on the correctness proofs of this typechecker, which are expected to be relatively easy: they mostly follow the inference rules.

Connecting it. We can extract this checker to OCaml and reuse the setup described in Section 1 to connect it with the reifier and easily derive a (partly certified) alternative checker for Coq's .vo files. Our initial tests indicate that it runs in reasonable time on medium-sized proof terms.

3 Internal Translations of CIC

We can now also use Template-Coq to write internal translations of CIC as advocated for in [4].

3.1 Global Framework

A internal translation is given by two functions defined by induction on the syntax: $[\cdot]$ which translates terms and $[\![\cdot]\!]$ which translates types. We define such functions in Coq, using the reified syntax of Template-Coq. Given a term t of type A we expect $[t]$ to be of type $[\![A]\!]$. As term and types share the same syntax in Coq, both functions have signature `term → term`.

Translating Constants. A constant c is translated, using an association table, by another constant c^t. If c has a body t (i.e. c is a definition) then c^t has $[t]$ as body. Otherwise c is an axiom, then c^t is either a definition (and the axiom is justified by the translation) or another axiom.

Global Contexts. Translations are also parameterized by a global context which records the declared inductive types and the declared constants. It is mandatory to get, for instance, arities of inductive types, types of constructors, ... For each inductive type, we need to specify how to translate its type, its constructors and its eliminator ; and enrich accordingly the global context and the translation table.

Under Specified Constructors. Coq's functions and applications are not fully typed. This can be problematic when we want to translate them. For instance, in the first translation of [4], we have:

$$[\lambda(x : A).t] := (\lambda(x : [\![A]\!]).[t],\ \text{true})$$

But in Coq the pair is typed and in fact we have:

`[fun (x:A) ⇒ t] := pair (∀ x:[[A]]. ??) 𝔹 (fun (x:[[A]]) ⇒ [t]) true`

and we can not recover the type ?? from the source term. Thankfully, Coq is designed so that type inference is decidable. That's why, provided we also kept track of the local context (the types of lambda variables crossed), we can use the type inference algorithm of Section 2 to recover the missing information.

Putting all this together our translation functions have signature:

`global_ctx → trans_table → local_ctx → term → term`

3.2 Coq Plugin

Given the two translation functions, we can extract them to ML code. We wrote a plugin that uses those extracted functions to let the user use the translation directly in Coq in the spirit of [4]. The plugin implements three commands:

- `Translate c` which derives the translation of a definition
- `Implement ax : A` which adds the axiom `ax : A` to the global environment provided the user is able to inhabit the translated type `[[A]]` (i.e. it switches to proof-mode).
- `Implement Existing c` which allows to define the translation of a constant for which the translation can not be automatically derived (inductive type, constructor or axiom).

Parametricity Anand and Morrisett [1] have already formalized a variant of the parametricity translation using Template-Coq, providing stronger theorems for free on propositions. We are also looking at re-implementing Lasson's plugin[2] for unmodified parametricity and giving its formal correctness proof.

3.3 Future Work

Our long term goal is to leverage this translation+plugin technique to extend the logical and computational power of Coq using, for instance, the forcing translation [6] or the weaning translation [8].

References

[1] A. Anand and G. Morrisett. Revisiting Parametricity: Inductives and Uniformity of Propositions. In *CoqPL'18*, Los Angeles, CA, USA, 2018.
[2] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *CoqPL*, Paris, France, 2017.
[3] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, November 1999.
[4] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *CPP'17, Paris, France*, pages 182–194. ACM, 2017.
[5] Yannick Forster and Fabian Kunze. Verified Extraction from Coq to a Lambda-Calculus. In *Coq Workshop 2016*, 2016.
[6] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. The definitional side of the forcing. In *LICS'16, New York, NY, USA*, pages 367–376, 2016.
[7] Gregory Michael Malecha. *Extensible Proof Engineering in Intensional Type Theory.* PhD thesis, Harvard University, 2014.
[8] Pierre-Marie Pédrot and Nicolas Tabareau. An effectful way to eliminate addiction to dependence. In *LICS'17, Reykjavik, Iceland*, pages 1–12, 2017.

[2] https://github.com/parametricity-coq/paramcoq